Maven Integration Test

jim stafford

Fall 2024 v2023-12-31: Built: 2024-11-19 21:38 EST

Table of Contents

1. Introduction . . . 1
   1.1. Goals . . . 1
   1.2. Objectives . . . 1
2. Maven Integration Test . . . 2
   2.1. Maven Integration Test Phases . . . 2
3. Maven Integration Test Plugins . . . 4
   3.1. Spring Boot Maven Plugin . . . 4
   3.2. Build Helper Maven Plugin . . . 7
   3.3. Failsafe Plugin . . . 8
4. Integration Test Client . . . 11
   4.1. JUnit @SpringBootTest . . . 11
   4.2. ClientTestConfiguration . . . 11
   4.3. username/password Credentials . . . 12
   4.4. ServerConfig . . . 12
   4.5. authnUrl URI . . . 13
   4.6. authUser RestTemplate . . . 13
   4.7. HTTP ClientHttpRequestFactory . . . 14
   4.8. JUnit @Test . . . 14
5. IDE Execution . . . 15
6. Maven Execution . . . 16
   6.1. Maven Verify . . . 16
7. Maven Configuration Reuse . . . 19
   7.1. it Maven profile . . . 19
   7.2. Failsafe HTTPS/IT Profile Example . . . 20
8. Summary . . . 24

Chapter 1. Introduction

Most of the application functionality to date within this course has been formally tested using unit tests and `@SpringBootTest` mechanisms executed within Maven Surefire plugin. This provided us a time-efficient code, test, and debug round trip — with possibly never leaving the IDE. However, the functionality of the stand-alone application itself has only been demonstrated using ad-hoc manual mechanisms (e.g., curl). In this lecture, we will look to test a fully assembled application using automated tests and the Maven Failsafe and other plugins.

Although generally slower to execute, Maven integration tests open up opportunities to implement automated tests that require starting and stopping external resources to simulate a more realistic runtime environment. These may be the application server or resources (e.g., database, JMS server) the application will communicate with.

1.1. Goals

You will learn:

- the need for Maven integration tests
- how to implement a Maven integration test
- how to define and activate a reusable Maven integration test setup and execution

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. implement a Maven Failsafe integration test in order to test a Spring Boot executable JAR running as a stand-alone process
2. dynamically generate test instance-specific property values in order to be able to run in a shared CI/CD environment
3. start server process(es) configured with test instance-specific property values
4. execute tests against server process(es)
5. stop server process(es) at the conclusion of a test
6. evaluate test results once server process(es) have been stopped
7. generalize Maven constructs into a conditional Maven parent profile
8. implement and trigger a Maven integration test using a Maven parent profile

Chapter 2. Maven Integration Test

Since we are getting close to real deployments, and we have hit unit integration tests pretty hard, I wanted to demonstrate a true integration test with the Maven integration test capabilities. Although this type of test takes more time to set up and execute, it places your application in a deployed simulation that can be tested locally before actual deployment. In this and follow-on chapters, we will use various realizations of the application and will start here with a simple Spring Boot Java application — absent of any backend resources.

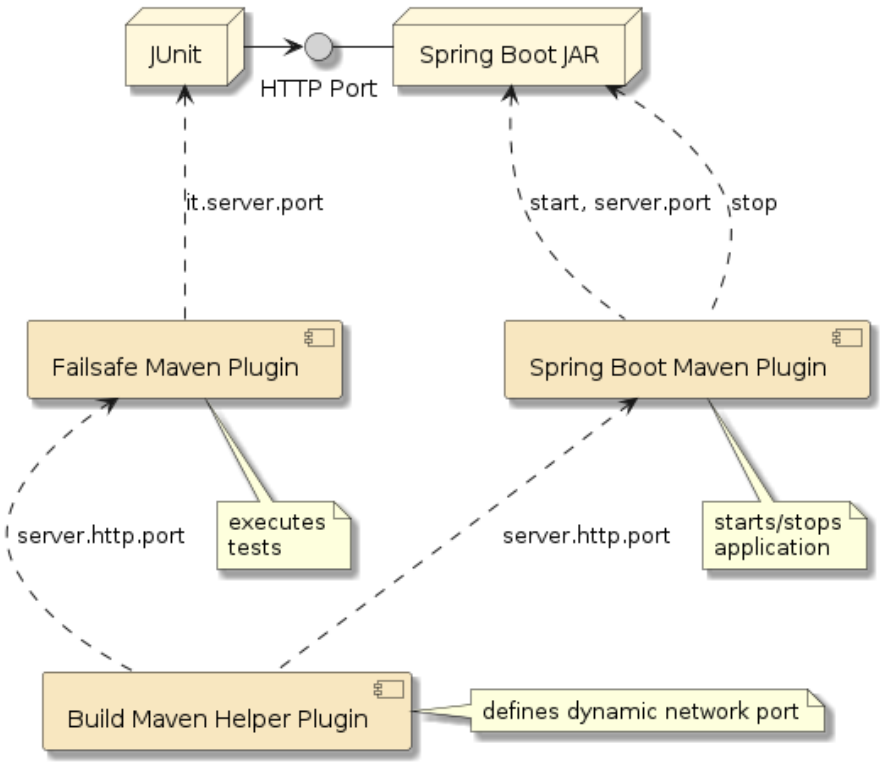

Figure 1. Maven Integration Test

A Maven integration test is very similar to the other Web API unit integration tests you are used to seeing in this course. The primary difference is that there are no server-side components in the JUnit Spring context. All the server-side components are in a separate executable (coming from the same or dependency module). The following diagram shows the participants that directly help to implement the integration test.

This will be accomplished with the aid of the Maven Failsafe, Spring Boot, and Build Maven Helper plugins.

With that said, we will still want to be able to execute integration tests like this within the IDE (i.e., start long-running server(s) and execute JUnit tests within the IDE during active test development). Therefore, expect some setup aspects to support both IDE-based and Maven-based integration testing setup in the paragraphs that follow.

2.1. Maven Integration Test Phases

Maven executes integration tests using four (4) phases

- `pre-integration-test` - start resources
- `integration-test` - execute tests
- `post-integration-test` - stop resources
- `verify` - evaluate/assert test results

These four (4) phases execute after the `test` phase (immediately after the `package` phase) and before the `install` phase. We will make use of three (3) plugins to perform that work within Maven:

- spring-boot-maven-plugin - used to start and stop the server-side Spring Boot process
- build-maven-helper-plugin - used to allocate a random network port for server
- maven-failsafe-plugin - used to run the JUnit JVM with the tests — passing in the port# — and verifying/asserting the results.

Chapter 3. Maven Integration Test Plugins

3.1. Spring Boot Maven Plugin

The `spring-boot-maven-plugin` will be configured with at least two (2) additional executions to support Maven integration testing.

1. package - (existing) builds the Spring Boot Executable JAR (bootexec)
2. **pre-integration-test** - (new) start our Spring Boot application under test
3. **post-integration-test** - (new) stop our Spring Boot application under test

The following snippet just shows the outer shell of the plugin declaration.

spring-boot-maven-plugin Outer Shell

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
    ... ①
  </executions>
</plugin>
```

① define goals, configuration, and phases for the plugin to perform

3.1.1. spring-boot:repackage Goal

We have already been using the `spring-boot-maven-plugin` to build an executable JAR from a Java JAR. The `build-app` execution was defined to run the `repackage` goal during the Maven `package` phase in ejava-build-parent/pom.xml to produce our Spring Boot Executable. The global configuration element is used to define properties that are applied to all executions. Each execution has a configuration element to define execution-specific configuration.

Pre-existing Use of Spring Boot Maven Plugin

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration> ①
        <attach>${spring-boot.attach}</attach> ③
        <classifier>${spring-boot.classifier}</classifier> ②
    </configuration>
    <executions>
      <execution>
        <id>build-app</id>
        <phase>package</phase> ④
        <goals>
          <goal>repackage</goal> ④
```

```
        </goals>
      </execution>
    </executions>
</plugin>
```

① global configuration applies to all executions

② Executable JAR will get a unique suffix identified by `classifier`

③ Executable JAR will not be installed or deployed in any Maven local or remote repository when `false`

④ `spring-boot-maven-plugin:repackage` goal executed during the Maven `package` phase with global configuration

3.1.2. spring-boot:start Goal

The next step is to start the built Spring Boot Executable JAR during the `pre-integration` phase — to be ready for integration tests.

The following snippet shows an example of how to configure a follow-on execution of the Spring Boot Maven Plugin to `start` the server in the background (versus a blocking `run`). The execution is configured to supply a Spring Boot `server.port` property with the HTTP port to use. We will define the port separately using the Maven `server.http.port` property at build time. The client-side `@SpringBootTest` will also need this port value for the client(s) in the integration tests.

SpringBoot pre-integration-test Execution

```
<execution>
  <id>pre-integration-test</id> ①
  <phase>pre-integration-test</phase> ②
  <goals>
    <goal>start</goal> ③
  </goals>
  <configuration>
    <skip>${skipITs}</skip> ④
    <arguments> ⑤
      <argument>--server.port=${server.http.port}</argument>
    </arguments>
  </configuration>
</execution>
```

① each execution must have a unique ID when there is more than one

② this execution will be tied to the `pre-integration-test` phase

③ this execution will `start` the server in the background

④ `-DskipITs=true` will deactivate this execution

⑤ `--server.port` is being assigned at runtime and used by server for HTTP listen port

skipITs support

> Most plugins offer a `skip` option to bypass a configured execution and sometimes map that to a Maven property that can be expressed on the command line. Failsafe maps their property to `skipITs`. By mapping the Maven `skipITs` property to the plugin's `skip` configuration element, we can inform related plugins to do nothing. This allows one to run the Maven `install` phase without requiring integration tests to run and pass.

The above execution phase has the same impact as if we launched the JAR manually with `--server.port`. This allows multiple IT tests to run concurrently without colliding on network port number. It also permits the use of a well-known/fixed value for use with IDE-based testing.

Manual Start Commands

```
$ java -jar target/failsafe-it-example-*-SNAPSHOT-bootexec.jar
Tomcat started on port(s): 8080 (https) with context path ''①

$ java -jar target/failsafe-it-example-*-SNAPSHOT-bootexec.jar --server.port=7712 ②
Tomcat started on port(s): 7712 (http) with context path '' ②
```

① Spring Boot using well-known default port#

② Spring Boot using runtime `server.port` property to override port to use

As expected, the Spring Boot Maven Plugin has first-class support for many common Spring Boot settings. For example:

- arguments - as just demonstrated
- environment variables
- JVM Arguments
- profiles
- etc.

You are not restricted to just command line arguments.

3.1.3. spring-boot:stop Goal

Running the tests is outside the scope of the Spring Boot Maven Plugin. The next time we will need this plugin is to shut down the server during the `post-integration-phase` — after the integration tests have completed.

The following snippet shows the Spring Boot Maven Plugin being used to `stop` a running server.

SpringBoot post-integration-test Execution

```
<execution>
  <id>post-integration-test</id> ①
  <phase>post-integration-test</phase> ②
  <goals>
    <goal>stop</goal> ③
```

```
  </goals>
  <configuration>
    <skip>${skipITs}</skip> ④
  </configuration>
</execution>
```

① each execution must have a unique ID

② this execution will be tied to the `post-integration-test` phase

③ this execution will `stop` the running server

④ `-DskipITs=true` will deactivate this execution

At this point the Spring Boot Maven Plugin is ready to do what we need as long as we define the Maven `server.http.port` property value to use. We can define that statically with a Maven pom.xml property, a `-Dserver.http.port= ####` on the Maven command line, or dynamically with the Maven Build Helper Plugin.

3.2. Build Helper Maven Plugin

The `build-helper-maven-plugin` contains various utilities that are helpful to create a repeatable, portable build. The snippet below shows the outer shell of the plugin declaration.

build-helper-maven-plugin Outer Shell

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>build-helper-maven-plugin</artifactId>
  <executions>
...
  </executions>
</plugin>
```

We will populate the declaration with one execution to identify the available network port.

3.2.1. build-helper:reserve-network-port Goal

We will run the Build Helper Maven Plugin early in the lifecycle — during the `process-resources` phase so that we have it early enough to use for unit tests (`test` phase) as well as integration tests (`pre-integration-test` phase when resources are started).

We are using the `reserve-network-port` goal to select one or more random and available HTTP port numbers at build-time and assign each to a provided Maven property (using the `portName` property). The port number, in this case, is assigned to the `server.http.port` Maven property. The `server.http.port` property was shown being used as input to the Spring Boot Maven Plugin earlier.

build-helper:reserve-network-port

```
<execution>
  <id>reserve-network-port</id>
```

```
    <phase>process-resources</phase> ①
    <goals>
      <goal>reserve-network-port</goal> ②
    </goals>
    <configuration>
      <portNames> ③
        <portName>server.http.port</portName>
      </portNames>
    </configuration>
  </execution>
```

① execute during the `process-resources` Maven phase — which is well before `pre-integration-test`

② execute the `reserve-network-port` goal of the plugin

③ assigned the identified port to the Maven `server.http.port` property

The snippet below shows an example of the `reserve-network-port` identifying an available network port.

Example build-helper:reserve-network-port Build-time Execution

```
[INFO] --- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
@ failsafe-it-example ---
[INFO] Reserved port 60066 for server.http.port
```

3.3. Failsafe Plugin

The Failsafe plugin is used to execute the JUnit tests. It is a near duplicate of its sibling Surefire plugin, but targeted to operate in the Maven `integration-test` phase. The following snippet shows the outer shell of the Failsafe declaration.

maven-failsafe-plugin Outer Shell

```
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <executions>
        ...
    </executions>
  </plugin>
```

3.3.1. failsafe:integration-test Goal

The `failsafe:integration-test` goal executes the JUnit tests and automatically binds to the Maven `integration-test` phase. This execution is separate from the server started by the Spring Boot Maven Plugin but must have some common configuration in order to communicate. In the snippet below, we are adding the `integration-test` goal to Failsafe and configuring the plugin to launch the JUnit tests with an `it.server.port` property.

Failsafe integration-test Phase

```
<execution>
    <id>integration-test</id>
    <goals> ①
      <goal>integration-test</goal>
    </goals>
    <configuration>
      <systemPropertyVariables> ②
        <it.server.port>${server.http.port}</it.server.port>
      </systemPropertyVariables>
    </configuration>
</execution>
```

① activate integration-test goal — automatically binds to `integration-test` phase

② add a `-Dit.server.port=${server.http.port}` system property to the execution

it.server.port used in ServerConfig used by Client

`it.server.port` is used to populate the `ServerConfig` `@ConfigurationProperties` component within the JUnit client.

```
@Bean
@ConfigurationProperties("it.server")
public ServerConfig itServerConfig() {
    return new ServerConfig();
}
```

3.3.2. failsafe:verify Goal

The snippet below shows the final phase for Failsafe. After the integration resources have been taken down, the only thing left is to assert the results. This is performed by the `verify` Failsafe goal, which automatically binds to the Maven `verify` phase. This pass/fail assertion is delayed by a few Maven phases so that the build does not fail while integration resources are still running.

Failsafe verify Phase

```
  <execution>
    <id>verify</id>
    <goals> ①
      <goal>verify</goal>
    </goals>
  </execution>
```

① activate verify goal — automatically binds to `verify` phase

verify Goal Must Be Declared to Report Pass/Fail to Maven Build

If the `verify` goal is not wired into the declaration — the setup, tests, and tear down

will occur, but the build will not evaluate and report the results to Maven. The integration tests will appear to always pass even if you see test errors reported in the `integration-test` output.

Chapter 4. Integration Test Client

With the Maven aspects addressed, let's take a look at any subtle changes we need to make to JUnit tests running within Failsafe.

4.1. JUnit @SpringBootTest

We start with a familiar looking JUnit test and `@SpringBootTest`. We can still leverage a Spring Context, however, there is no application or server-side resources in the Spring context. The application has its Spring Context started by the Spring Boot Maven Plugin. This Spring Context is for the tests running within the Failsafe Plugin.

```
@SpringBootTest(classes=ClientTestConfiguration.class, ①
        webEnvironment = SpringBootTest.WebEnvironment.NONE) ②
public class FailsafeRestTemplateIT {
    @Autowired
    private RestTemplate authnUser;
    @Autowired
    private URI authnUrl;
```

① no application class in this integration test. Everything is server-side.

② have only a client-side web environment. No listen port necessary

4.2. ClientTestConfiguration

This trimmed down `@Configuration` class is all that is needed for JUnit test to be a client of a remote process. The `@SpringBootTest` will demand to have a `@SpringBootConfiguration` (`@SpringBootApplication` is a `@SpringBootConfiguration`) to initialize the application hosting the test. So, we will assign that annotation to our test's configuration.

We also need to add `@EnableAutoConfiguration` (normally supplied by `@SpringBootApplication`) to enable injection of external resources like `RestTemplateBuilder`.

JUnit Client SpringBootConfiguration

```
@SpringBootConfiguration(proxyBeanMethods = false) ①
@EnableAutoConfiguration ②
@Slf4j
public class ClientTestConfiguration {
    ...
    @Bean
    @ConfigurationProperties("it.server") ③
    public ServerConfig itServerConfig() { ...
    @Bean
    public URI authnUrl(ServerConfig serverConfig) { ...④
    @Bean
    public RestTemplate authnUser(RestTemplateBuilder builder,...⑤
```

```
    ...
```

① there must be 1 `@SpringBootConfiguration` and must be supplied when running without a `@SpringBootApplication`

② must enable AutoConfiguration to trigger `RestTemplateBuilder` and other automatic resources

③ inject properties for test (e.g., `it.server.port`) from Failsafe Maven Plugin

④ injectable baseUrl of remote server

⑤ injectable `RestTemplate` with authentication and HTTP filters applied

Since Maven integration tests have no `RANDOM_PORT` and no late `@LocalServerPort` injection, bean factories for components that depend on the server port do not require `@Lazy` instantiation.

4.3. username/password Credentials

The following shows more of the `@SpringBootConfiguration` with the username and password credentials being injected using values from the properties provided a test properties file. In this test's case — they should always be provided. Therefore, no default String is defined.

username/password Credentials

```
public class ClientTestConfiguration {
    @Value("${spring.security.user.name}")
    private String username;
    @Value("${spring.security.user.password}")
    private String password;
```

4.4. ServerConfig

The following shows the primary purpose for `ServerConfig` as a `@ConfigurationProperties` class with flexible prefix. In this particular case, it is being instructed to read in all properties with prefix "it.server" and instantiate a ServerConfig. The Failsafe Maven Plugin is configured to supply the random port number using the `it.server.port` property.

```
@Bean
@ConfigurationProperties("it.server")
public ServerConfig itServerConfig() {
    return new ServerConfig();
}
```

The URL scheme will default to "http" and the port will default to "8080" unless a property override (`it.server.scheme`, `it.server.host`, and `it.server.port`) is supplied. The resulting value will be injected into the `@SpringBootConfiguration` class.

4.5. authnUrl URI

Since we don't have the late-injected `@LocalServerPort` for the web-server and our `ServerConfig` values are all known before starting the Spring Context, we no longer need `@Lazy` instantiation. The following shows the `baseUrl` from `ServerConfig` being used to construct a URL for "/api/authn/hello".

Building baseUrl from Injected ServerConfig

```
import org.springframework.web.util.UriComponentsBuilder;
...
@Bean
public URI authnUrl(ServerConfig serverConfig) {
    URI baseUrl = serverConfig.getBaseUrl();
    return UriComponentsBuilder.fromUri(baseUrl).path("/api/authn/hello").build()
.toUri();
}
```

4.6. authUser RestTemplate

We are going to create separate `RestTemplate` components, fully configured to communicate and authenticate a particular way. This one and only `RestTemplate` for our example will be constructed by the `authnUser` `@Bean` factory.

By no surprise, `authnUser()` `@Bean` factory is adding a `BasicAuthenticationInterceptor` containing the injected username and password to a new `RestTemplate` for use in the test. The injected `ClientHttpRequestFactory` will take care of the HTTP/HTTPS details.

authnUser RestTemplate

```
import org.springframework.http.client.BufferingClientHttpRequestFactory;
import org.springframework.http.client.ClientHttpRequestFactory;
import org.springframework.http.client.support.BasicAuthenticationInterceptor;
import org.springframework.web.client.RestTemplate;
...
@Bean
public RestTemplate authnUser(RestTemplateBuilder builder,
                              ClientHttpRequestFactory requestFactory) {
    RestTemplate restTemplate = builder.requestFactory(
        //used to read the streams twice -- so we can use the logging filter below
        ()->new BufferingClientHttpRequestFactory(requestFactory))

        .interceptors(new BasicAuthenticationInterceptor(username, password), ①

                new RestTemplateLoggingFilter())
        .build();
    return restTemplate;
}
```

① adding a `ClientHttpRequestInterceptor` filter, supplied by Spring to supply an HTTP BASIC

Authentication header with username and password

4.7. HTTP ClientHttpRequestFactory

The `ClientHttpRequestFactory` is supplied as a `SimpleClientHttpRequestFactory` using injection to separate the underlying transport (TLS vs. non-TLS) from the HTTP configuration.

HTTP-based ClientHttpRequestFactory

```
import org.springframework.http.client.ClientHttpRequestFactory;
import org.springframework.http.client.SimpleClientHttpRequestFactory;
...
@Bean
public ClientHttpRequestFactory httpsRequestFactory() {
    return new SimpleClientHttpRequestFactory(); ①
}
```

① an HTTP/(non-TLS)-based `ClientHttpRequestFactory`

4.8. JUnit @Test

The core parts of the JUnit test are pretty basic once we have the HTTP/Authn-enabled `RestTemplate` and baseUrl injected. From here it is just a normal test, except that the activity under test is remote on the server side.

```
public class FailsafeRestTemplateIT {
    @Autowired ①
    private RestTemplate authnUser;
    @Autowired ②
    private URI authnUrl;

    @Test
    public void user_can_call_authenticated() {
        //given a URL to an endpoint that accepts only authenticated calls
        URI url = UriComponentsBuilder.fromUri(authnUrl)
            .queryParam("name", "jim").build().toUri();

        //when called with an authenticated identity
        ResponseEntity<String> response = authnUser.getForEntity(url, String.class);

        //then expected results returned
        then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        then(response.getBody()).isEqualTo("hello, jim");
    }
}
```

① `RestTemplate` with authentication and HTTP aspects addressed using filters

② `authnUrl` built from `ServerConfig` and injected into test

Chapter 5. IDE Execution

JUnit tests, like all software, take a while to write and the development is more efficiently tested within the IDE. The IDE can run the JUnit test shown here just like any other test. The IDE does not care about class naming conventions. The IDE will run any class method annotated with `@Test` as a test. However, the application must be manually started (and restarted after changes to `src/main`).

There are at least two (2) ways to start the server when working with the JUnit test in the IDE. In all cases, the port should default to something well-known like `8080`.

- use the IDE to run the application class
- use the Spring Boot Maven Plugin to run the application class. The `server.port` will default to `8080`.

  Using Spring Boot Maven Plugin to Run Application

  ```
  $ mvn spring-boot:run
  ...
  [INFO] --- spring-boot-maven-plugin:3.3.2:run (default-cli) @ failsafe-it-example
  ---
  ...
  Tomcat started on port 8080 (http) with context path '/'
  Started FailsafeExampleApp in 1.63 seconds (process running for 1.862)
  ```

  Supply an extra `spring-boot.run.arguments` system property to pass in any custom arguments, like port#.

  Using Spring Boot Maven Plugin to Run Application with Custom Port#

  ```
  $ mvn spring-boot:run -Dspring-boot.run.arguments='--server.port=7777' ①
  ...
  [INFO] --- spring-boot-maven-plugin:3.3.2:run (default-cli) @ failsafe-it-example
  ---
  ...
  Tomcat started on port(s): 7777 (http) with context path '/'
  Started FailsafeExampleApp in 2.845 seconds (process running for 3.16)
  ```

  ① use override to identify desired `server.port`; otherwise use 8080

Once the application under test by the JUnit test is running, you can then execute individual integration tests annotated with `@Test`, entire test cases, or entire packages of tests using the IDE.

Once the integration tests are working as planned, we can automate our work using Maven and the plugins we configured earlier.

Chapter 6. Maven Execution

Maven can be used to automatically run integration tests by automating all external resource needs.

6.1. Maven Verify

When we execute `mvn verify` (with an option to add `clean`), we see the port being determined and assigned to the `server.http.port` Maven property.

Starting Maven Build

```
$ mvn verify
...①
- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port) @
failsafe-it-example ---
[INFO] Reserved port 52024 for server.http.port
...②
- maven-surefire-plugin:3.3.2:test (default-test) @ failsafe-it-example ---
...
- spring-boot-maven-plugin:3.3.2:repackage (build-app) @ failsafe-it-example ---
...③
- spring-boot-maven-plugin:3.3.2:start (pre-integration-test) @ failsafe-it-example
---
...
```

① the port identified by build-helper-maven-plugin as `52024`

② Surefire tests firing at an earlier `test` phase

③ server starting in the `pre-integration-test` phase

mvn install also runs verify

The Maven `verify` goal is right before `install`. Calling `mvn clean install` is common and runs all Surefire tests and Failsafe integration tests.

6.1.1. Server Output

When the server starts, we can see that the default profile is active and Tomcat was assigned the `52024` port value from the build.

Server Output

```
FailsafeExampleApp#logStartupProfileInfo:634 No active profile set, falling back to 1
default profile: "default" ①
TomcatWebServer#initialize:108 Tomcat initialized with port(s): 52024 (http)②
```

① no profile has been activated on the server

② server HTTP port assigned to `52024`

6.1.2. JUnit Client Output

When the JUnit client starts, we can see that the baseURL contains `http` and the dynamically assigned port `52024`.

JUnit Client Output

```
FailsafeRestTemplateIT#logStartupProfileInfo:640 No active profile set, falling back
to 1 default profile: "default" ①
ClientTestConfiguration#authnUrl:42 baseUrl=http://localhost:52024 ②
FailsafeRestTemplateIT#setUp:30 baseUrl=http://localhost:52024/api/authn/hello
```

① no profile is active in JUnit client

② baseUrl is assigned `http` and port `52024`, with the latter dynamically assigned at build-time

6.1.3. JUnit Test DEBUG

There is some DEBUG logged during the activity of the test(s).

Message Exchange

```
GET /api/authn/hello?name=jim, headers=[accept:"text/plain, application/json,
application/xml, text/xml, application/*+json, application/*+xml, */*",
authorization:"Basic dXNlcjpwYXNzd29yZA==", user-agent:"masked",
host:"localhost:52024", connection:"keep-alive"]]
```

6.1.4. Failsafe Test Results

Test results are reported.

Failsafe Test Results

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.168 s -- in
info.ejava.examples.svc.https.FailsafeRestTemplateIT
Results:
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

6.1.5. Server is Stopped

Server is stopped.

Server is Stopped

```
- spring-boot-maven-plugin:3.3.2:stop (post-integration-test) @ failsafe-it-example
---
 Stopping application...
15:44:26.510 RMI TCP Connection(4)-127.0.0.1  INFO
```

```
XBeanRegistrar$SpringApplicationAdmin#shutdown:159 Application shutdown requested.
```

6.1.6. Test Results Asserted

Test results are asserted.

Overall Test Results

```
[INFO] --- maven-failsafe-plugin:3.3.2:verify (verify) @ failsafe-it-example ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

Chapter 7. Maven Configuration Reuse

The plugins can often become a mouthful to completely define, and we would like to conditionally reuse that definition and activate them in sibling modules using Maven integration tests. We can define them in the parent pom and make them optionally activated within Maven profiles and activated using various activation conditions.

- JDK version
- OS
- Property presence and/or value
- Packaging
- File presence
- manual activation (`-P profile_name`) or deactivation (`-P !profile_name`)

One or more conditions can be defined. Since Maven 3.2.2, all conditions have been required to be satisfied to activate the profile.

Maven Profile Activation Criteria is ANDed

Maven profile activation lacks expressive predicate logic. There is no option for `OR`.

Maven Profile is Separate from Spring Profile

Both Spring and Maven have the concept of a "profile". The two are similar in concept but totally independent of one another. Activating a Maven profile does not inherently activate a Spring profile and vice versa.

Bash Requires bang("!") to be Escaped

Linux/bash shells require a bang("!") character to be escaped.

```
$ mvn clean install -P \!it
```

7.1. it Maven profile

The following snippet shows the Shell of an `it` Maven profile being defined to activate when the file `application-it.properties` appears in the `src/test/resources` directory. We can place the plugin definitions from above into the profile. We can also include other properties specific to the profile.

```
<profiles>
    <profile>
        <id>it</id> ①
        <activation>
            <file> ②
                <exists>${basedir}/src/test/resources/application-
it.properties</exists>
```

```
            </file>
        </activation>
        <properties>
            ...
        </properties>
        <build>
            <pluginManagement>
                <plugins>
                    ...
                </plugins>
            </pluginManagement>

            <plugins> <!-- activate the configuration -->
                ...
            </plugins>
        </build>
    </profile>
</profiles>
```

① defining profile named `it`

② profile automatically activates when the specific file exists

7.2. Failsafe HTTPS/IT Profile Example

I have added a second variant in a separate Maven Failsafe example module (`failsafe-https-example`) that leverages the Maven `it` profile in `ejava-build-parent`. The Maven profile activation only cares that the `src/test/resources/application-it.properties` file exists. The contents of the file is of no concern to the Maven `it` profile itself. The contents of `application-it.properties` is only of concern to the Spring `it` profile (when activated with `@ActiveProfiles`).

7.2.1. it.properties Maven Trigger File

Presence of File Activates Profile

```
<profile>
    <id>it</id>
    <activation>
        <file>
            <exists>${basedir}/src/test/resources/application-it.properties</exists>
        </file>
    </activation>

|-- pom.xml
`-- src
    ...
    `-- test
        ...
        `-- resources
            `-- application-it.properties ①
```

① file presence (empty or not) triggers Maven profile

7.2.2. Augmenting Parent Definition

This `failsafe-https-example` example uses a Spring `https` profile for the application and requires the Spring Boot Maven Plugin to supply that Spring profile when the application is launched. We can augment the parent Maven plugin definition by specifying the plugin and execution ID with augmentations.

Specific options for `configuration` can be found in the Spring Boot Maven Plugin run goal documentation

Augmenting Plugin Definition with Profile Configuration

```
<build>
    <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <executions>
            <execution>
                <id>pre-integration-test</id>
                <configuration> ①
                    <profiles>https</profiles>
                </configuration>
            </execution>
        </executions>
    </plugin>
```

① child pom augmenting parent configuration definition

Additional Parent Plugin Override/Customization

We have some control over override versus append using `combine.children` and `combine.self` attributes of the configuration.

7.2.3. Integration Test Properties

The later `failsafe-https-example` will require a few test properties to be defined. We can use the `application-it.properties` Maven profile activation file trigger to define them. You will recognize them from the HTTPS Security lecture.

application-it.properties

```
it.server.scheme=https
#must match self-signed values in application-https.properties
it.server.trust-store=${server.ssl.key-store}
it.server.trust-store-password=${server.ssl.key-store-password}
#used in IDE, overridden from command line during failsafe tests
it.server.port=8443
```

Same Basic HTTPS Setup Demonstrated with Surefire in the HTTPS Security Chapter

The example also requires the client Spring Context to be capable of establishing a TLS connection. This is exactly as shown in the HTTPS Security chapter and will not be shown here. See the `failsafe-https-example` project for the complete example these excerpts were taken from.

7.2.4. Activating Test Profiles

The JUnit test now activates two profiles: `https` and `it`.

- the `https` profile is used to define the server communication properties that `application-it.properties` uses for templated property expansion. The client does not use these directly.
- the `it` profile is used to define the client communication properties.

Since the client property values are defined in terms of server properties, both profiles must be active to bring in both sources.

Activate Server HTTPS Properties and Client Profile

```
@SpringBootTest(classes=ClientTestConfiguration.class,
        webEnvironment = SpringBootTest.WebEnvironment.NONE)
//https profile defines server properties used here by client
//it profile defines client-specific properties
@ActiveProfiles({"https","it"}) ① ②
public class HttpsRestTemplateIT {
```

① `https` profile activates server properties for client IT test context to copy

② `it` profile defines client IT test properties templated using `https` defined properties

7.2.5. Example HTTPS Failsafe Test Execution

The following `mvn verify` output shows the complete build for the `failsafe-https-example` project which leverages the reusable `it` profile form the build parent.

Summary of Full Maven verify Build

```
$ mvn verify
- maven-resources-plugin:3.3.1:resources (default-resources)
- build-helper-maven-plugin:3.4.0:reserve-network-port (reserve-network-port)
Reserved port 59871 for server.http.port
- maven-compiler-plugin:3.11.0:compile (default-compile) @ https-hello-example ---
- maven-compiler-plugin:3.11.0:testCompile (default-testCompile) @ https-hello-example
---
- maven-surefire-plugin:3.1.2:test (default-test) @ https-hello-example ---
- maven-jar-plugin:3.3.0:jar (default-jar) @ https-hello-example ---
- maven-source-plugin:3.3.0:jar (attach-sources) @ https-hello-example ---
- spring-boot-maven-plugin:3.3.2:start (pre-integration-test) @ https-hello-example
---
...
```

```
HttpsExampleApp#logStartupProfileInfo:640 The following 1 profile is active: "https"
TomcatWebServer#initialize:108 Tomcat initialized with port(s): 59871 (https)
StandardService#log:173 Starting service [Tomcat]
...
- maven-failsafe-plugin:3.1.2:integration-test (integration-test)
...
Running info.ejava.examples.svc.https.HttpsRestTemplateIT
...
HttpsRestTemplateIT#logStartupProfileInfo:640 The following 2 profiles are active:
"https", "it"
ClientTestConfiguration#authnUrl:55 baseUrl=https://localhost:59871
...

[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
- spring-boot-maven-plugin:3.3.2:stop (post-integration-test)
[INFO] Stopping application...
- maven-failsafe-plugin:3.1.2:verify (verify)
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

Chapter 8. Summary

In this module, we learned:

- the need for Maven integration tests
- implement a Maven Failsafe integration test in order to test a Spring Boot executable JAR running as a stand-alone process
  - dynamically generate test instance-specific property values in order to be able to run in a shared CI/CD environment
  - start server process(es) configured with test instance-specific property values
  - execute tests against server process(es)
  - stop server process(es) at the conclusion of a test
  - evaluate test results once server process(es) have been stopped
- how to define reusable Maven integration test setup and execution
  - generalize Maven constructs into a conditional Maven parent profile
  - implement and trigger a Maven integration test using a Maven parent profile